CRISPFLOW, A Structured Program Design Flowcharter

R. C. Tausworthe and K. C. Landon
DSN Data Systems Section

This article describes the design, prototype implementation, and use of an automated tool, called CRISPFLOW, for generating design flowcharts of structured programs. The user of this tool provides a file of English-like textual statements which directs the drawing of the flowcharts. Layout, symbol sizing, location of text, scaling, and other routine tasks are performed automatically.

I. Introduction

CRISPFLOW is a software design aid which automatically generates ANSI-standard (and DSN Standard) flowcharts in response to directives in a structured program design language given to a computer. The name derives from Control-Restricted Instructions for Structured Program FLOWcharting. Because CRISPFLOW creates flowcharts at design time (from a program design language), rather than at implementation time (from a computer language), it serves as a tool which encourages top-down development of software, relieving the designer of the hand-drawn flowchart burden. Flowcharts are easily available and economically produced, edited, and redrawn during the design phase of a software development. The charts produced are graphic representations of the specifications for coding, not after-the-fact documentation of what was coded.

The user of CRISPFLOW creates a file of statements grouped into modules; each such statement represents one of the symbols on a flowchart to be drawn. The syntax of CRISPFLOW statements is taken from a subset of CRISP (Ref. 1), except that English phrases can be used within the syntax, as well as computer-language constructs, since no code is being generated from CRISPFLOW statements. Each module flowcharted occupies one standard page.

CRISPFLOW produces flowcharts for a canonic subset of structures most commonly used for structured programming. Less common structures, which include the extended IF form, the indexed looping structure, paranormal exits, and other components of CRISP described in Ref. 1 are not included in the CRISPFLOW processor as of this report.

The CRISPFLOW processor reads a module from a user file and converts this to an internal representation which describes the flowchart symbols to be drawn, the sizes of these symbols, where these symbols are to appear on a page, and what text is to be printed in and around each symbol. The user needs only to prepare a structured source program as described in the next Section of this report. All topological considerations – layout, symbol sizes, text locating, scaling, etc. – are performed automatically by the CRISPFLOW processor.

The CRISPFLOW processor, as of this report, operates as an MBASIC™ program on the Univac 1108 which computes the placement of the flowchart symbols on the page and generates the points for plotting. When this is complete, a

FORTRAN subprogram draws the chart on either CALCOMP or COM plotter, the latter on microfilm or microfiche.

II. The CRISPFLOW Language

As described in Reference 1, the CRISP concept is one whereby a control restrictive syntax is superimposed on some base language of the user's choice. Whenever the base language is an abbreviated form of technical English, the constructs form what is known as a procedure (or program) design language. This CRISPFLOW prototype recognizes certain of the CRISP constructs and generates structured flowcharts in response. These constructs hereafter will be referred to as the CRISPFLOW language.

This report will not detail this language, but, will rather illustrate its capabilities. Further information on the CRISPFLOW language and processor is available from the authors.

CRISPFLOW permits the flowcharting of PROGRAM, PROCEDURE(TO), and SUBROUTINE modules containing IFTHENELSE, CASE, LOOP-REPEAT-IF, and LOOP-WHILE-REPEAT flowchart structures. Each CRISPFLOW language statement begins with a keyword which signifies the flowchart symbol to be drawn. Each CRISPFLOW statement results in a specific flowchart response.

A. Base Language Statements

Text which does not begin with one of the recognized CRISPFLOW keywords is transferred directly onto the chart, inserted into a standard process flowchart symbol (a rectangle). For example, the text

BASE LANGUAGE STATEMENT

in the CRISPFLOW input file creates the flowchart image shown in Figure 1a because the word "BASE" is not a recognized keyword.

CRISPFLOW breaks the text string at spaces as needed to fit, and centers the resulting substrings within the rectangle. If the substring cannot fit inside the default-size rectangle, CRISPFLOW selects a rectangle sufficiently large to contain all of the text and then centers the text inside it.

B. CRISPFLOW Comment Annotation

Comments in the CRISPFLOW source images are denoted by the delimiters "<*" and "*>", which enclose the comment. Comments may appear in any part of any statement. When a comment appears in a statement, CRISPFLOW usually discards it, and the comment does not appear on the flowchart. For example, the input statement

BASE <*COMMENT*>LANGUAGE STATEMENT

again results in the same flowchart image as in Figure 1a.

The comment text is not always discarded, however. In particular, the comment field in PROGRAM, TO, PROCEDURE, SUBROUTINE, DO, CALL, and CALLX statements *do* appear on the flowchart in a manner described later.

C. Box Numbers and Cross-Reference Annotation

Statements in CRISPFLOW have optional additional fields available for annotating the ANSI standard box number and cross-reference notations (see Ref. 2) on each of the flowchart symbols. For example, the form

.5/X1 BASE LANGUAGE STATEMENT

results in the flowchart response shown in Figure 1b. The text string between "." and "/" is placed at the upper right of the flowchart symbol, while the text string following "/" up to the first space is placed at the upper left of the flowchart symbol; the entire text message beyond (if not beginning with a keyword) is placed inside the box as described previously.

D. Source Cosmetics

Certain optional characters preceding the CRISP keyword or base language statement are considered "cosmetic", and are ignored by CRISPFLOW. These characters are: "!", ":", "-", ">", "_", and ".", when "." is not in column 1. Cosmetic characters may be used to enhance readability of the input file and such usage will be illustrated in subsequent examples. For the present, observe that

.5/X1 !:: ... BASE LANGUAGE STATEMENT

results in the same flowchart image as shown in Figure 1b, previously described.

E. Module Delimiters

A *module,* as used here, is a set of statements that CRISPFLOW flowcharts as a single page. Each module flowcharted must begin and end with one of the following pairs of delimiter keywords:

PROGRAM	PROCEDURE	TO	SUBROUTINE
ENDPROGRAM	ENDPROCEDURE	ENDTO	ENDSUBROUTINE

A sample short module and its flowchart are shown in Figure 2. In the upper right-hand corner, CRISPFLOW has entered the text following MOD#, then, on the next line, the text field following the keyword and up to, but not including "MOD#"; next, the comment (date) text, and finally; "PAGE OF ". Any of the first three of these strings may be absent. The fourth always appears. CRISPFLOW also supplies a signature box for the designer, coder, and auditor; below this appears the date the flowchart was drawn.

CRISPFLOW draws the same entry symbol in response to the module-header keywords PROGRAM, PROCEDURE, TO, and SUBROUTINE, and the exit symbol in response to ENDPROGRAM, ENDPROCEDURE, ENDTO, and ENDSUBROUTINE. Text following MOD# on the module header is drawn above and left of the entry symbol, and text following the keyword is printed within the entry symbol. CRISPFLOW prints "RETURN" in the exit symbol of SUBROUTINE modules.

F. Striped Symbols

The ANSI standard convention (Ref. 2) for indicating that a more detailed description or representation of a function is to be found elsewhere is by means of "striping" the chart symbol horizontally or vertically. CRISPFLOW uses statements having the keywords DO, CALL, and CALLX for this purpose. The text following the keyword appears at the top of each such symbol, and the comment field, if there is one, appears in the lower part of the symbol. Multiple comment fields are merged together. For example, the statements

```
.8      DO PARSE <*CONVERT MODULE TO A TREE &
           STRUCTURE.*>
.4/S2   CALL INVMTX<*INVERT A MATRIX.*>
.12/X3  CALLX PRTLIN<*PRINT A LINE*>&
           <*ON THE TERMINAL.*>
```

cause CRISPFLOW to draw Figures 3a, 3b, and 3c, respectively. CRISPFLOW stripes DO and CALL statements horizontally, and CALLX statements vertically. Note the use of the ampersand at the end of a line to continue the statement.

The CRISP DO statement is used to reference a procedure flowchart by the same name elsewhere in the current documentation set; the CALL statement represents a linkage to a subroutine documented elsewhere in the current documentation set. The CALLX form denotes reference to a procedure or subroutine *not* in the current documentation set, but documented at the place cited by the cross-reference identifier (X3 above). CALLX statements, for example, are commonly used to denote invocations of library subroutines.

G. The IFTHENELSE Structure

Binary decisions are plotted using the IFTHENELSE structure, which requires a block of statements of the form illustrated by

```
.4    IF (DELTA<0)
.5    :    Z=X+Y
      :->(ELSE)
.6    :    Z=X-Y
      : .ENDIF
```

Note the use of cosmetics to enhance readability of the CRISPFLOW source. The result is shown in Figure 4.

On the flowchart, CRISPFLOW places the condition under test in the diamond followed by a question mark; "THEN" and "ELSE" clauses are drawn to the left and right of the diamond, respectively, and are labelled "T" and "F". The user may have as many statements as desired in either the THEN or the ELSE clauses. CRISPFLOW recognizes either ELSE or (ELSE) as the beginning of the ELSE-clause. The ENDIF statement is required to terminate the IF-block.

H. Looping Structures

CRISPFLOW provides two of the more common iterative structures: both are bounded within LOOP. . .REPEAT statements in the source module, as illustrated in the following two examples, which produce Figures 5 and 6, respectively.

```
PROCEDURE: NEWTON'S METHOD <* 11 Aug 77*> &
              MOD# 1
      <* THIS ITERATIVE PROCEDURE COMPUTES THE
      <* ROOT OF SOME UNSPECIFIED FUNCTION OF X
      <* STARTING WITH AN APPROXIMATION OF THE
      <* ROOT. A LIMIT IS PLACED ON THE NUMBER
      <* OF ITERATIONS TO FORCE LOOP TERMINA-
      <* TION IF CONVERGENCE FAILS OR IS TOO
      <* SLOW.
.1    X=BEST GUESS OF ROOT, LIMIT=100, &
        T=THRESHOLD
.2    LOOP
.3    ! DELTA=FUNCTION(X)/DERIVATIVE(X)
.4    ! X=X-DELTA, LIMIT=LIMIT-1, &
      !    ENOUGH=(ABS(DELTA)>T AND LIMIT>0)
.5    !_REPEAT IF (NOT ENOUGH)
      ENDPROCEDURE
```

```
SUBROUTINE: SEARCH  <*29 AUG 76*>  MOD# S29
      <* ARGUMENTS ARE VAL: INTEGER,
      <* FOUND: BOOLEAN.
      <* SEARCH THROUGH INTEGER ARRAY FOR
      <* VAL, AND SET
      <* FOUND ACCORDINGLY.
.1    SELECT INITIAL PORTION TO BE SEARCHED
.2    LOOP WHILE (PORTION SIZE>1)
.3    ! REDUCE PORTION TO BE SEARCHED
      !_REPEAT
.4    EXAMINE CHOSEN PART, AND SET FOUND
      ENDSUBROUTINE
```

I. The CASE Structure

When branching in a program may take more than two alternate paths, the CASE structure may be used. The format of the CASE structure is illustrated in the next example, which is a module that invokes one of three different subroutines depending on the discriminant of a quadratic equation ($b^2 - 4ac$):

```
TO FINDROOT  <*11 AUG 77*>  MOD# 1.5.7.3
.1    DISCRIMINANT =B**2-4*A*C
.2    CASE (DISCRIMINANT)
      :->(=0)
.3    :   CALL ONEROOT <*PRINT SINGLE VALUE*>
      :->(>0)
.4/T7:    CALL REALROOTS <*PRINT 2 REAL &
      :       VALUES*>
      :->(<0)
.5/N3:    CALL COMPROOTS <*PRINT 2 COMPLEX &
      :       VALUES*>
      : . .ENDCASES
      ENDTO
```

The decision condition follows the keyword CASE in parentheses, parenthesized case labels signal the beginnings of clauses, and the keyword ENDCASES terminates the structure. The flowcharter draws the decision-text string in a diamond, draws the appropriate number of outcome flowpaths below the decision symbol, and labels the flowpaths with the various outcomes. Figure 7 shows the flowcharted result.

J. Nested Structures

CRISPFLOW automatically configures substructures to be flowcharted within structures. The following SAMPLE procedure illustrates the nesting of structures in both source and flowchart (Figure 8) forms:

```
PROCEDURE: SAMPLE <*18 SEPT 75*> MOD# 1.3.5
      <* THIS SAMPLE MODULE DEMONSTRATES THE
      <* CRISPFLOW SYNTAX WITH A HYPOTHETICAL
      <* MESSAGE TRANSMISSION SYSTEM. STATE-
      <* MENTS DENOTED BY ST1 THROUGH ST4 ARE
      <* UNSPECIFIED HERE, AND THE READ ROUTINE
      <* IS EXTERNAL TO THE SET OF DOCUMENTA-
      <* TION FOR WHICH THE FLOWCHART IS BEING
      <* PRODUCED.
.1    IF (UNALLOCATED)
.2       CASE (MODE)
            (1)
.3             ST1
            (2)
.4             ST2
.5             ST3
            (3)
.6             ST4
         ENDCASES
.7/S1    CALL OPEN (MODE) <*CHANNEL IS MODE &
            NUMBER.*>
      (ELSE)
.8       LOOP WHILE (AVAILABLE)
.9/XS2      CALLX READ (CHR) <*READ CHARACTER*>
.10/S3      CALL WRITE(CHR)<*WRITE CHARACTER*>
            REPEAT
      ENDIF
.11   DO CLOSE<*MESSAGE SENT*>
.12   DO RELEASE<*DISCONNECT CHANNEL*>
      ENDPROCEDURE
```

Note that comments after the procedure declaration do not transfer to the chart. In this case, the flowchart is less explanatory than the source module, and should therefore be accompanied by supplementary explanatory narrative.

III. The CRISPFLOW Processor Design

The top-level CRISPFLOW control structure repeats for each module in the source file the following three steps:

```
PARSE "the module into a tree"
LAYOUT "the nodes of the tree on a plotter page"
DRAW "the boxes and flowlines on the chart"
```

The parser scans each line of the source input stream to pick up "tokens" (Refs. 3, 4) and uses a bottom-up parse, based on the first "keyword" token of each source line, to construct an internal representation of the module's parse tree. The SAMPLE of Figure 8 generates the tree structure shown in Figure 9, for example. The phrase structure of the CRISP grammar is designed so that there exists a close correspondence between nodes in this tree and "super-boxes"

(discussed below) on the chart. Each node in the tree corresponds to a string of terminals in the grammar, and ultimately is also made to reference all the information needed to locate, draw, fill, and connect the box it represents.

The box-arrangement algorithm in the CRISPFLOW system, LAYOUT, is responsible for translating between the procedural and flowchart media in a manner which yields both elegant and homomorphic mappings between source statements and flowchart structures. The importance of the former criterion cannot be over-emphasized, for the utility of the system to potential users is greatly diminished if the boxes and flowlines on a chart are not arranged in a straightforward, well-formatted and easily-readable fashion. At the same time, however, economic and human engineering considerations dictate that such machine constraints as processing time and memory requirements not be ignored.

The algorithm herein adopted for LAYOUT is a compromise, therefore, between a "dynamic programming" approach and one which forestalls repetitive, brute-force search schemes to allow rapid chart layout with only modest memory demands. The "super-box" approach yields very well-arranged charts without attempting to pack boxes as densely (or as "cleverly"), as might be expected.

The layout routine scans the parser-generated tree in a postorder, or bottom-up, traverse (Ref. 5), and for each node encountered, forms a transparent "super-box" which encloses the superboxes of all its (nested) subtrees. The size of each superbox is determined by the type of node (i.e., statement) to which it corresponds. The postorder walk permits the grammatical structure of the tree to be utilized as a "reduction system", or means to layout the chart in a single, comprehensive bottom-up pass through the tree. As the nodes at each level in the tree are scanned and "reduced" to super-boxes, they are used, in turn, to construct the super-boxes of the nodes at the next higher level. This procedure continues until the top-level statements of a module are combined into the super-box for the entire chart.

The DRAWing routine is activated after the layout tree traversal, and it first centers the super-box of the module on the plotter page and scales it down to fit, if necessary. It then performs a preorder, or top-down, tree scan, during which it establishes coordinates, draws the boxes, fills them with text, and connects them with flowlines, arrows, and collecting nodes.

A. Super-box Definitions

The figures of this section illustrate the super-box formats necessary to represent the flowchart symbols in this design, along with their structured conglomerates. Figure 10 shows the atomic boxes which are actually drawn in and annotated. Figure 11 illustrates how "then" and "else" sub-superboxes stand in relation to the superbox of the IFTHENELSE structure. Superboxes for LOOP, CASE and other structures are similarly defined.

In Figures 10 and 11, DH is a unit of horizontal spacing (normally about 3 mm) and DV is that for vertical spacing (normally about 5 mm). These allow for box separation, arrow heads, etc. SBH and SBW (possibly with distinguishing subscripts) are acronyms for Super-Box Height and Width, respectively, and are shown in Figure 11 with the equations used to compute them from the known dimensions of their constituent sub-super-boxes (either atomic or conglomerate). The coordinates of all chart symbols within a super-box are given relative to the axes of that super-box; the origin of a super-box is located at its top dead center (i.e., at the point bisecting its top edge).

Super-box outlines never intersect, and it is this property which makes LAYOUT as straightforward as it is. To be more precise, one super-box may enclose or be enclosed by others, or two super-boxes may share a common edge, but no two super-boxes may ever overlap. No super-box spacing is needed since the super-box which directly encloses a chart symbol includes sufficient space around its perimeter to give an eye-pleasing format. Thus, the layout algorithm simply "pushes" together the super boxes it "picks up" in a postorder walk according to the "blueprints" such as that in Figure 11. Once the conglomeration of boxes comprises a complete module, the chart origin and scale are determined to fit the chart on the page.

B. Data Structures

The two primary structures needed for the above-sketched algorithms are a binary tree (containing the parsed module descriptions) and a pushdown stack (to enable walks of the tree). Each node in the tree has identical structure, with sufficient fields to keep box number, cross-reference, type, size, x-y coordinates, and text information. Two pointer fields are included in addition, so that the module can be mapped onto a standard (leftmost descendant, sibling) binary representation (thus allowing a constant number of fields for each node packet), as shown in Figure 9. An entry in the stack consists of a pointer to a node in the tree.

C. The Tree

The attributes of each flowchart symbol are organized into the structure TREE. It is declared as a vector of records, and each record, representing one node, has the fields described below. The nodes are linked together into the tree structure by two pointer fields in each node, denoted TREE.BROTHER

and TREE.SON. These pointers permit flowchart representation via a binary mapping of the tree nesting levels.

The fields (not all applicable to each node class) within each node of the TREE are:

NAME	DESCRIPTION
CLASS	the type of node (e.g., IF, CASE,. . .).
SON, BROTHER	Pointers
BOXNUM, XREF	The box number and cross-reference identification strings for the chart symbol.
TEXT, COMMENT	The texts of the CRISP or base language statement and any comment on the same line.
SBW, SBH	The width and height of the super-box associated with this node.
W	The width of the chart symbol to which this node corresponds.
X,Y	The chart coordinates of this superbox.

IV. Operational Characteristics

The Univac 1108 current implementation of CRISPFLOW is coded in the MBASIC language (Refs. 6 and 7), and therefore the source files to direct plotting must be MBASIC processor compatible (standard data file format). Such files may be created using the MBASIC processor or the JPL text editor.

CALCOMP flowcharts are normal (21.6 X 28 cm or 8-1/2 X 11 inch) size but lack the drawing quality of the COM plotter charts. The COM plotter produces high-quality charts on either 35mm film or 105mm microfiche; 35mm output also is accomplished by a 23-cm (9-inch) photostat furnished automatically. Full-page reproductions are available then from the Central Reproductions Facility.

The cost of a flowchart, such as SAMPLE in Figure 8, is about $4-$6 at prime-shift computer rates; during weekends, the rate drops to about $.40-$.60 per chart. When the MBASIC batch compiler becomes available, however, these costs are expected to drop by a factor of 5 to 10. Recoding the CRISPFLOW processor in a non-interpretive language would also provide approximately the same operational cost savings.

V. Conclusion

The use of CRISPFLOW as a design tool brings the power of a computer-based system to relieve many of the objections often levied against flowcharting. Editing and updates are as easily accomplished on charts as they are on any other data files. The charts are accurately and quickly drawn, and displayed in excellent drafting quality. Further, due to the high level of the CRISPFLOW language and the 1-1 correspondence between statements and chart symbols, programmers themselves quickly learn to use the source versions of modules, and order the drawing of flowcharts only when they become satisfied with their designs and desire documentation of a more graphic form for others.

The processor described here is only a prototype; undoubtedly it will evolve and improve as users interact with it and suggest improvements. Usage will probably increase when an MBASIC compiler becomes available to reduce costs. Even in its current state, however, it is demonstrating the benefits of computer-based graphic documentation of computer programs.

References

1. Tausworthe, Robert C., *Standardized Development of Computer Software; Part 1, Methods,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977, Chapter 7.
2. "American National Standard Flowchart Symbols and Their Usage in Information Processing", American National Standards Institute, Inc., ANSI X3.5-1970, Sept. 1, 1970.
3. Donovan, John J., *Systems Programming,* McGraw-Hill, Inc., 1972, pp. 227-240.
4. Aho, A. V., and Ullman, J. D., *The Theory of Parsing, Compiling, and Translation,* Vols. I and II, Prentice-Hall, Inc., Englewood Cliffs, N.J.
5. Knuth, D., *Fundamental Algorithms,* Vol. I, Addison-Wesley Pub. Co., Reading, Mass., 1969.
6. "MBASIC, Vol. 1, Fundamentals," Jet Propulsion Laboratory, Pasadena, Ca., March 1973 (JPL internal document).
7. "MBASIC, Vol. 2, Appendices", Jet Propulsion Laboratory, Pasadena, Ca., Oct. 1973 (JPL internal document).

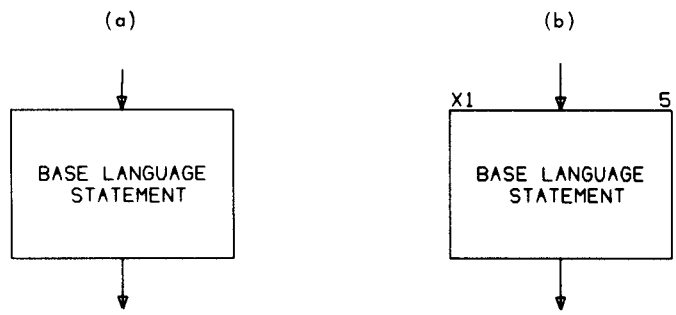

Fig. 1. Response to base language statements: (a) without box number or cross-reference annotations; (b) with annotations

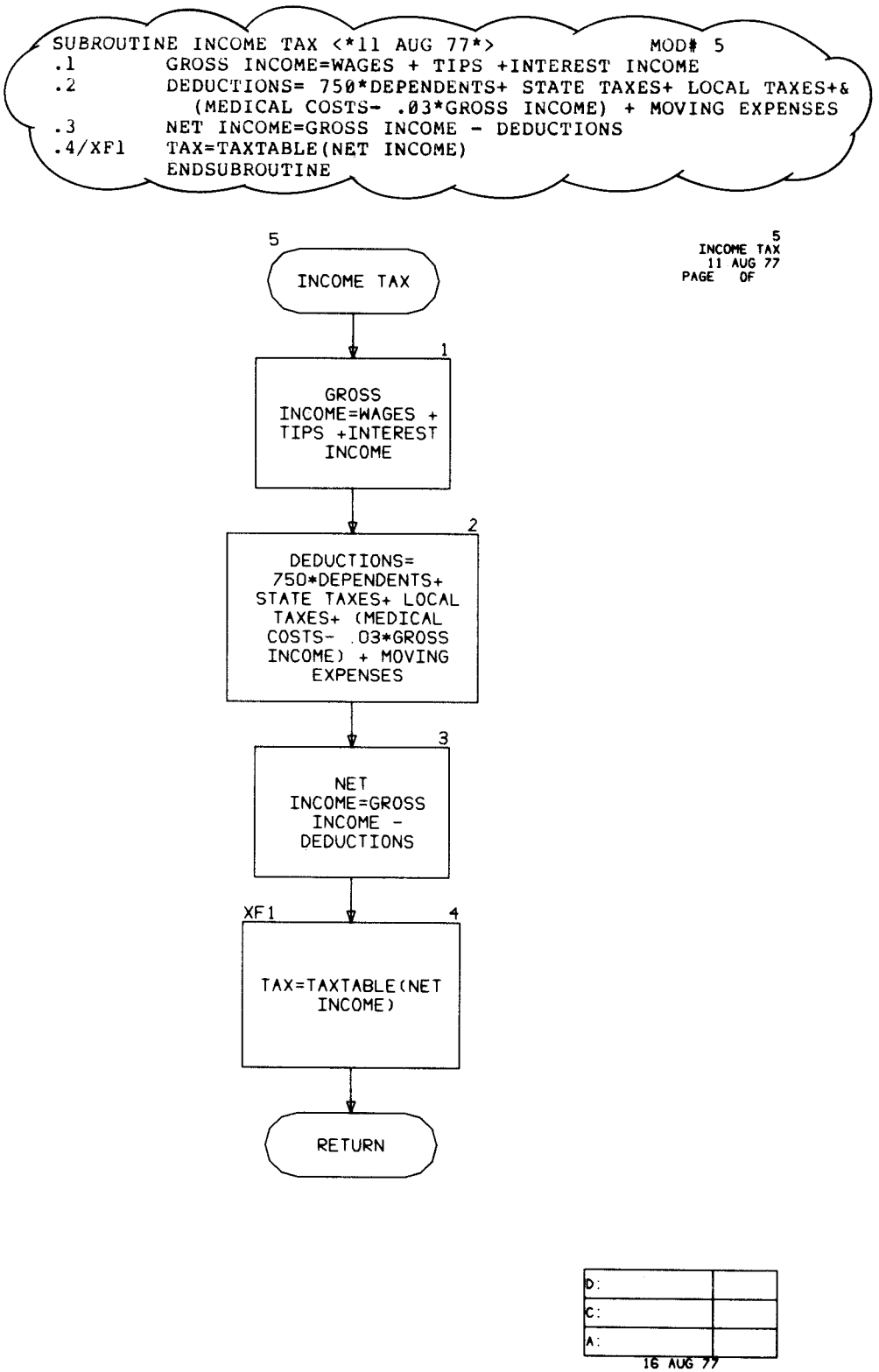

Fig. 2. A sample flowchart showing entry and exit symbols, plus other annotations supplied by CRISPFLOW for convenience in finding the chart among a set of charts on other documentation and for design control signatures

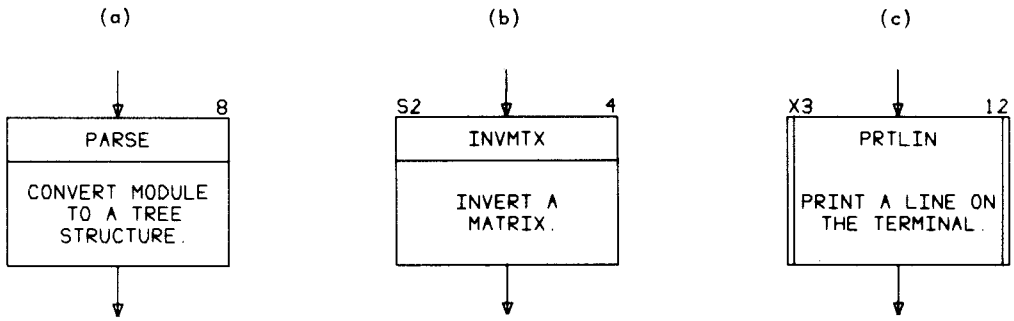

Fig. 3. Striped symbols used for displaying procedure, internal subroutine, and external subroutine documentation

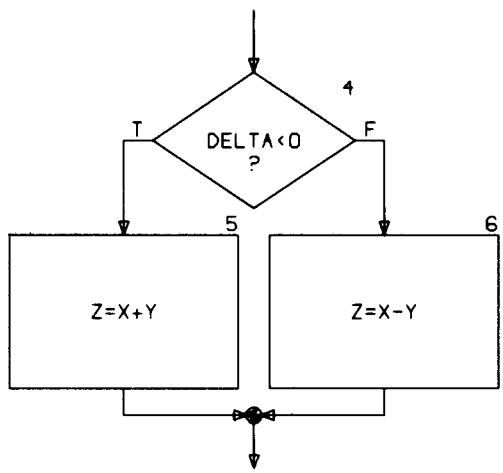

Fig. 4. Flowcharted response to an IFTHENELSE block of statements in CRISPFLOW

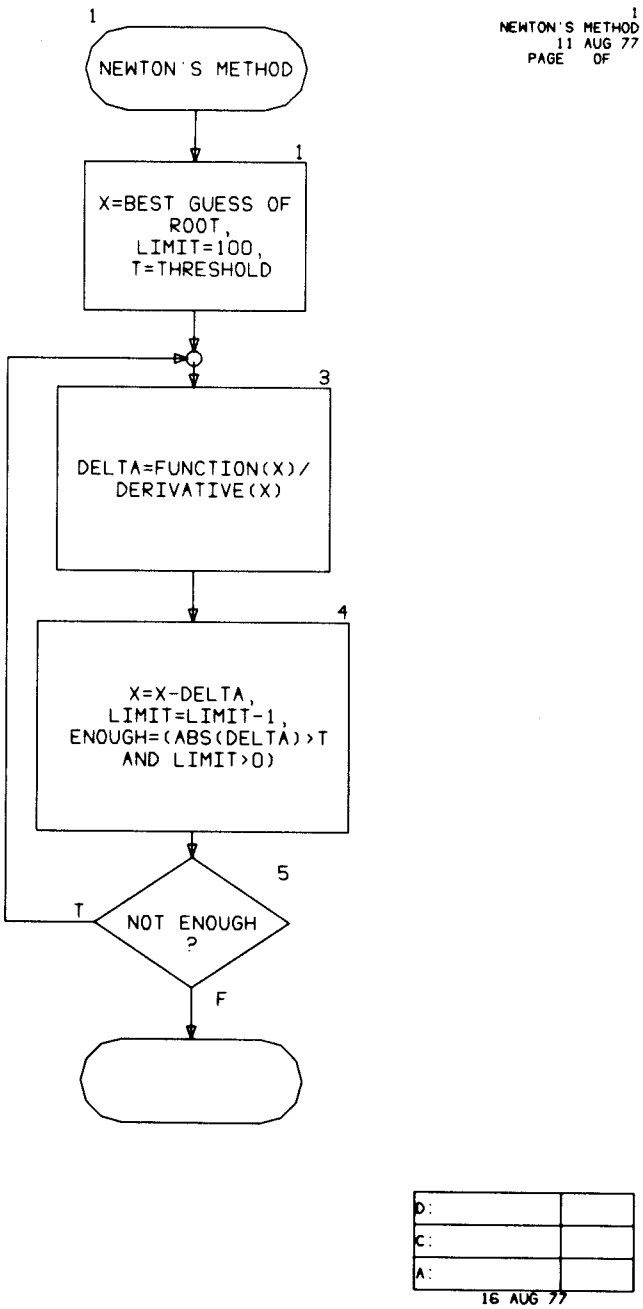

Fig. 5. Procedure module demonstrating LOOP . . REPEAT IF block of statements

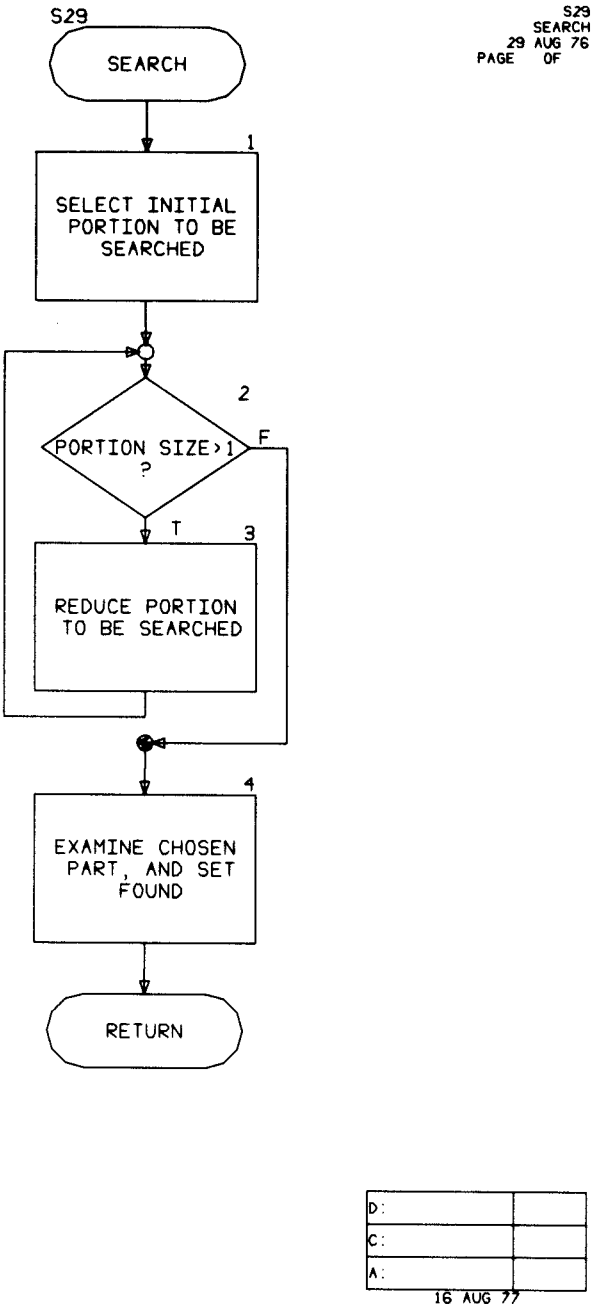

Fig. 6. Flowchart of a subroutine module illustrating a LOOP WHILE . . REPEAT block of statements

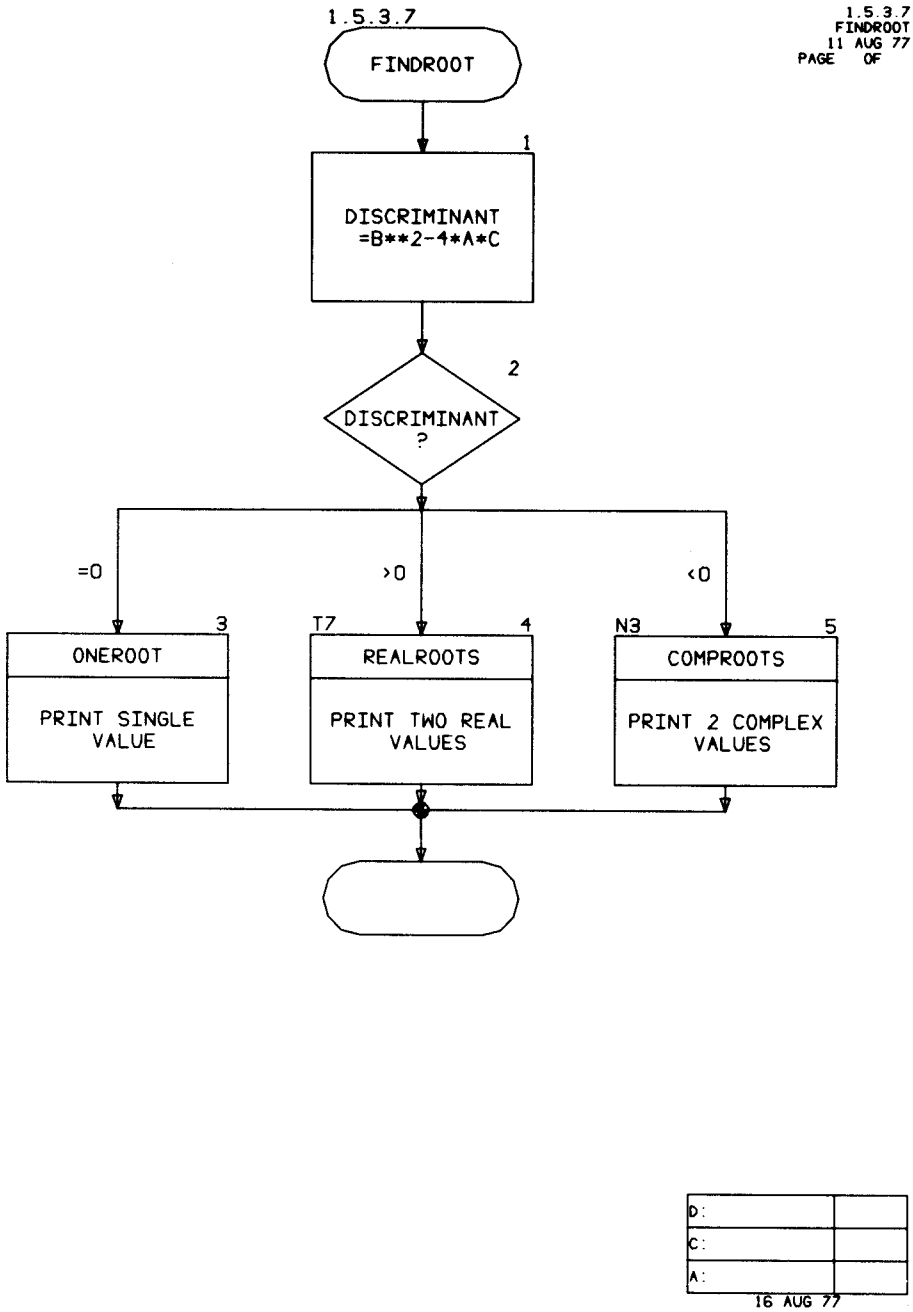

Fig. 7. Flowcharted result of a TO module containing a CASE structure

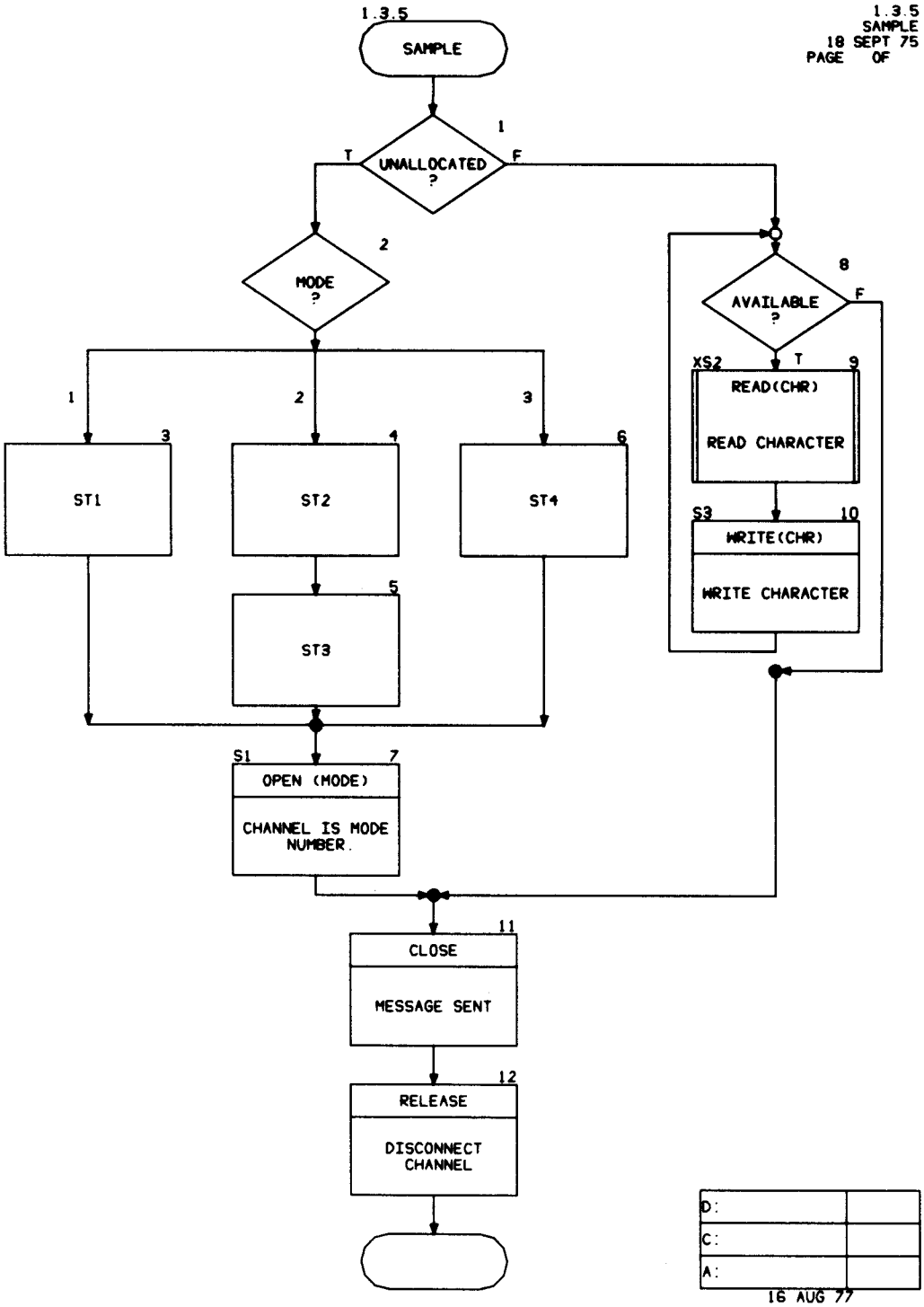

Fig. 8. SAMPLE procedure illustrating nesting of structures

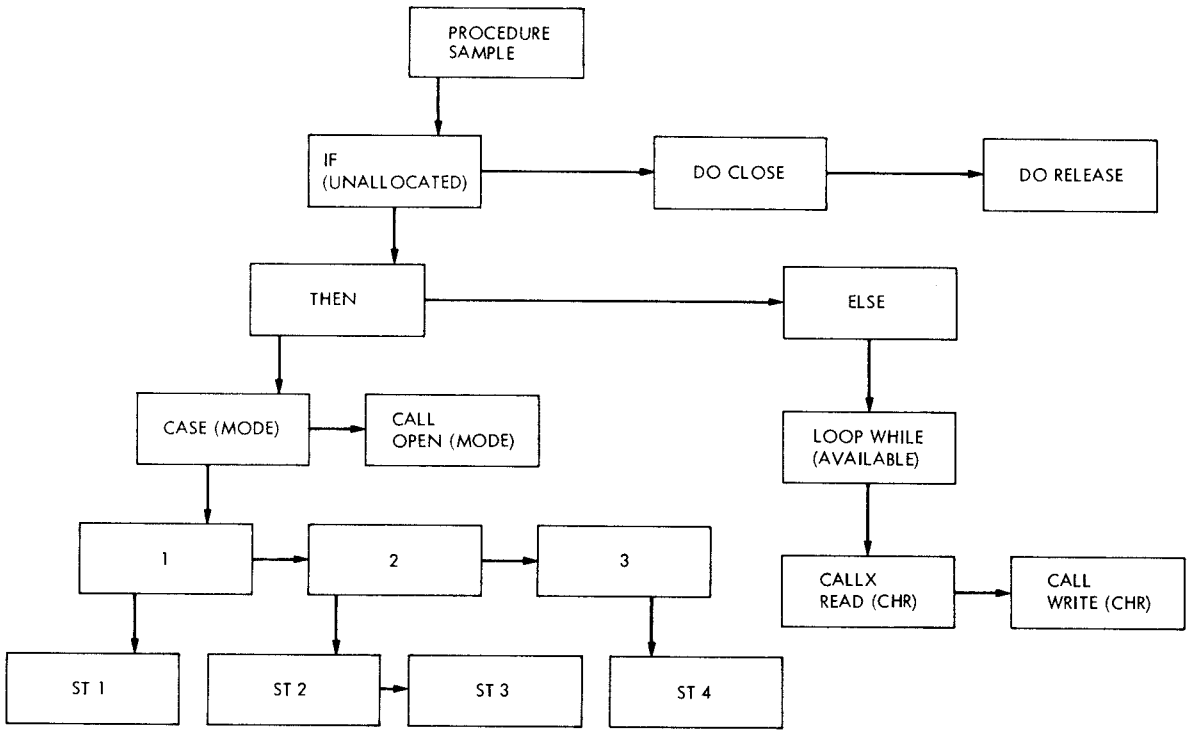

Fig. 9. The Binary Parse Tree of Procedure "SAMPLE". SON pointers exit the bottoms of boxes, BROTHER pointers leave the sides

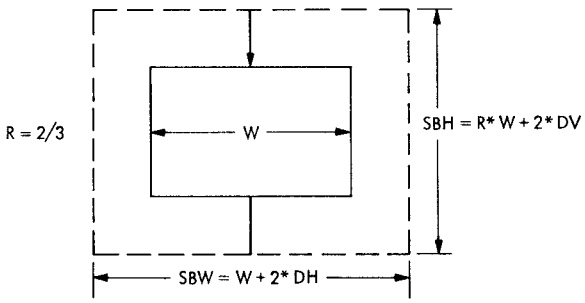

Fig. 10a. The standard rectangle for CALL, CALLX, DO, and target statements

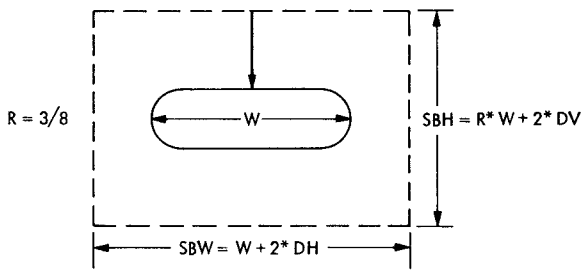

Fig. 10b. The oval for module exits

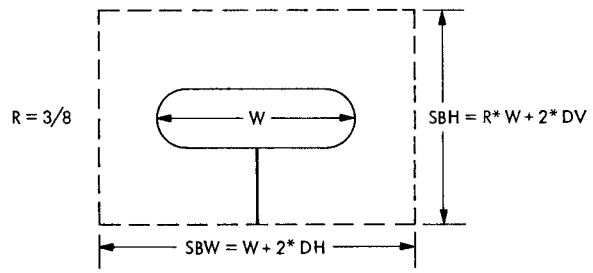

Fig. 10c. The oval for module entrances

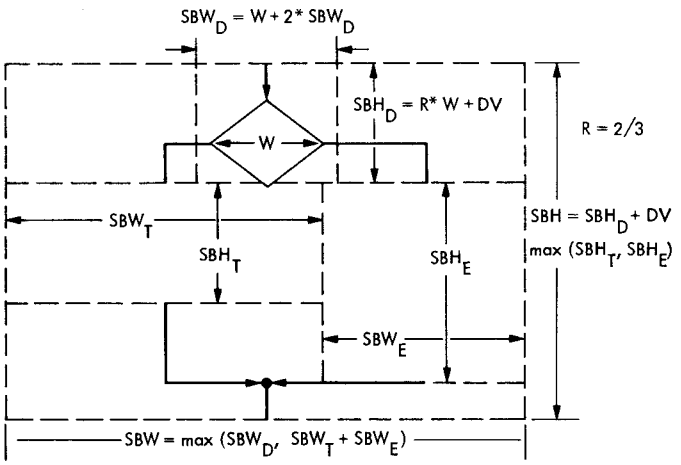

Fig. 11. The super-box of an IF structure with Decision super-box and Then and Else clause statement list super-boxes as constituent parts